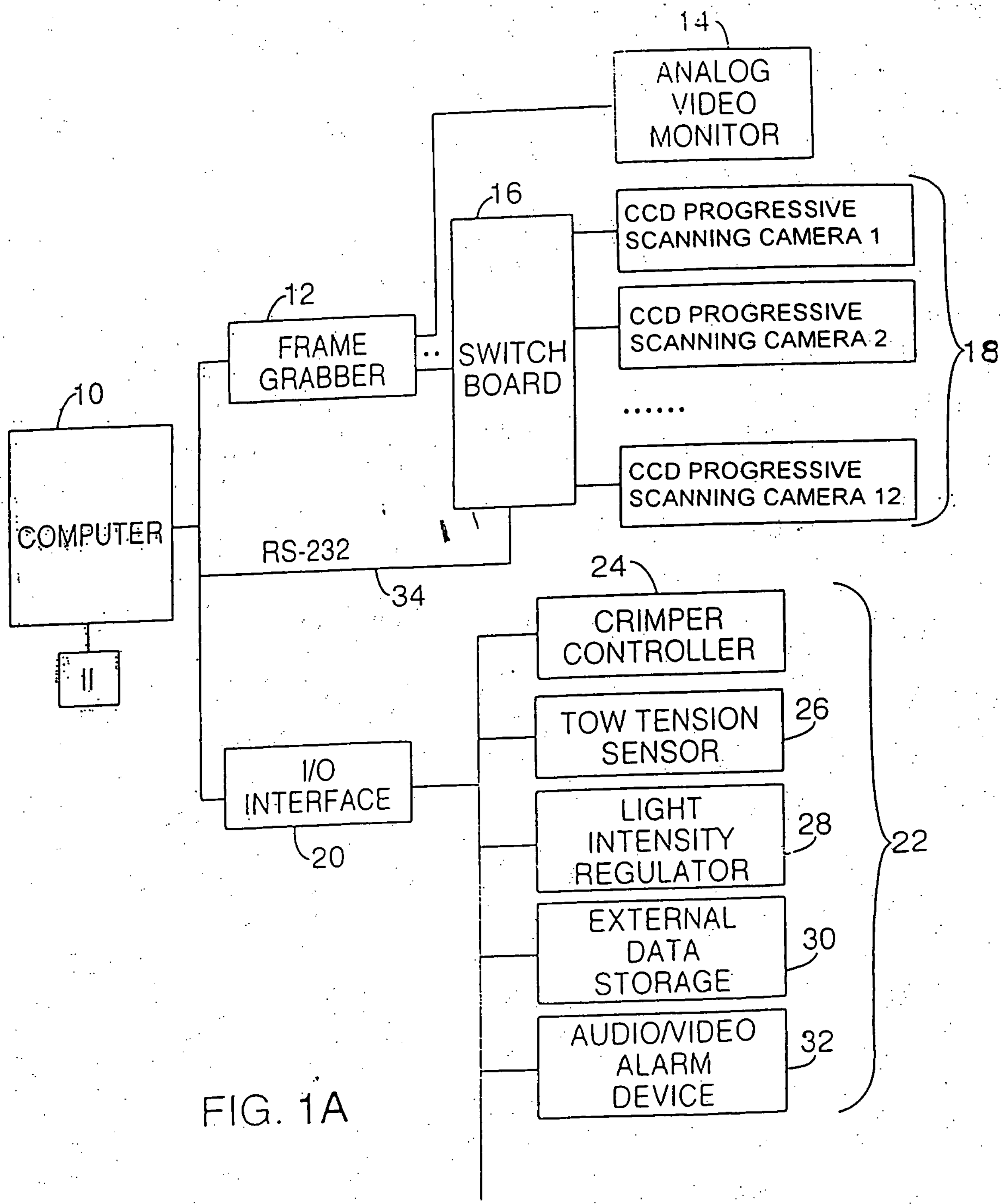

14
ANALOG VIDEO MONITOR
16
SWITCH BOARD
CCD PROGRESSIVE SCANNING CAMERA 1
CCD PROGRESSIVE SCANNING CAMERA 2
CCD PROGRESSIVE SCANNING CAMERA 12
18
12
FRAME GRABBER
10
COMPUTER
11
RS-232
34
24
CRIMPER CONTROLLER
TOW TENSION SENSOR
26
LIGHT INTENSITY REGULATOR
28
EXTERNAL DATA STORAGE
30
AUDIO/VIDEO ALARM DEVICE
32
22
I/O INTERFACE
20

FIG. 1A

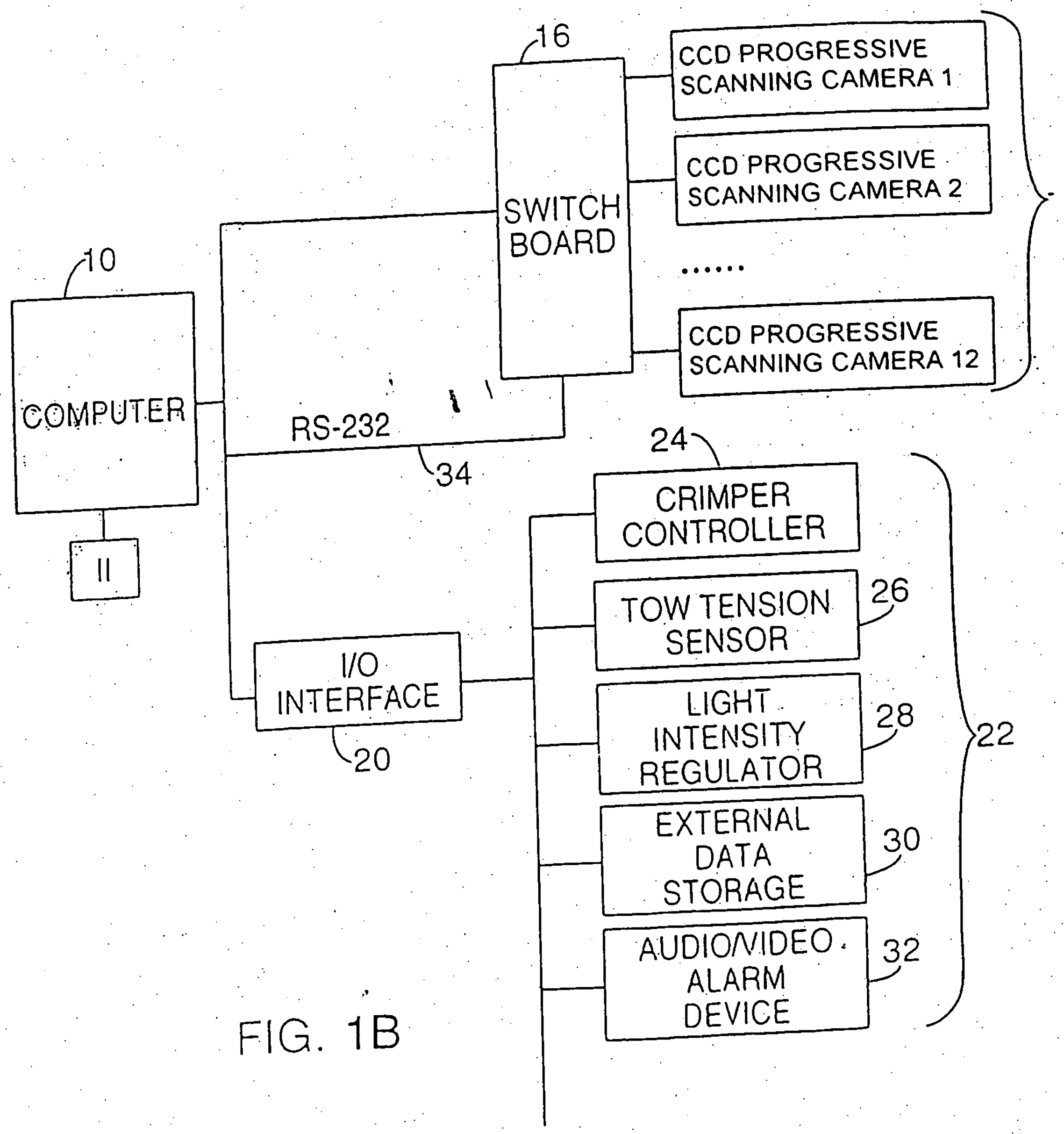

16
CCD PROGRESSIVE SCANNING CAMERA 1
CCD PROGRESSIVE SCANNING CAMERA 2
......
CCD PROGRESSIVE SCANNING CAMERA 12
SWITCH BOARD
10
COMPUTER
RS-232
34
11
24
CRIMPER CONTROLLER
TOW TENSION SENSOR
26
I/O INTERFACE
20
LIGHT INTENSITY REGULATOR
28
22
EXTERNAL DATA STORAGE
30
AUDIO/VIDEO ALARM DEVICE
32

FIG. 1B

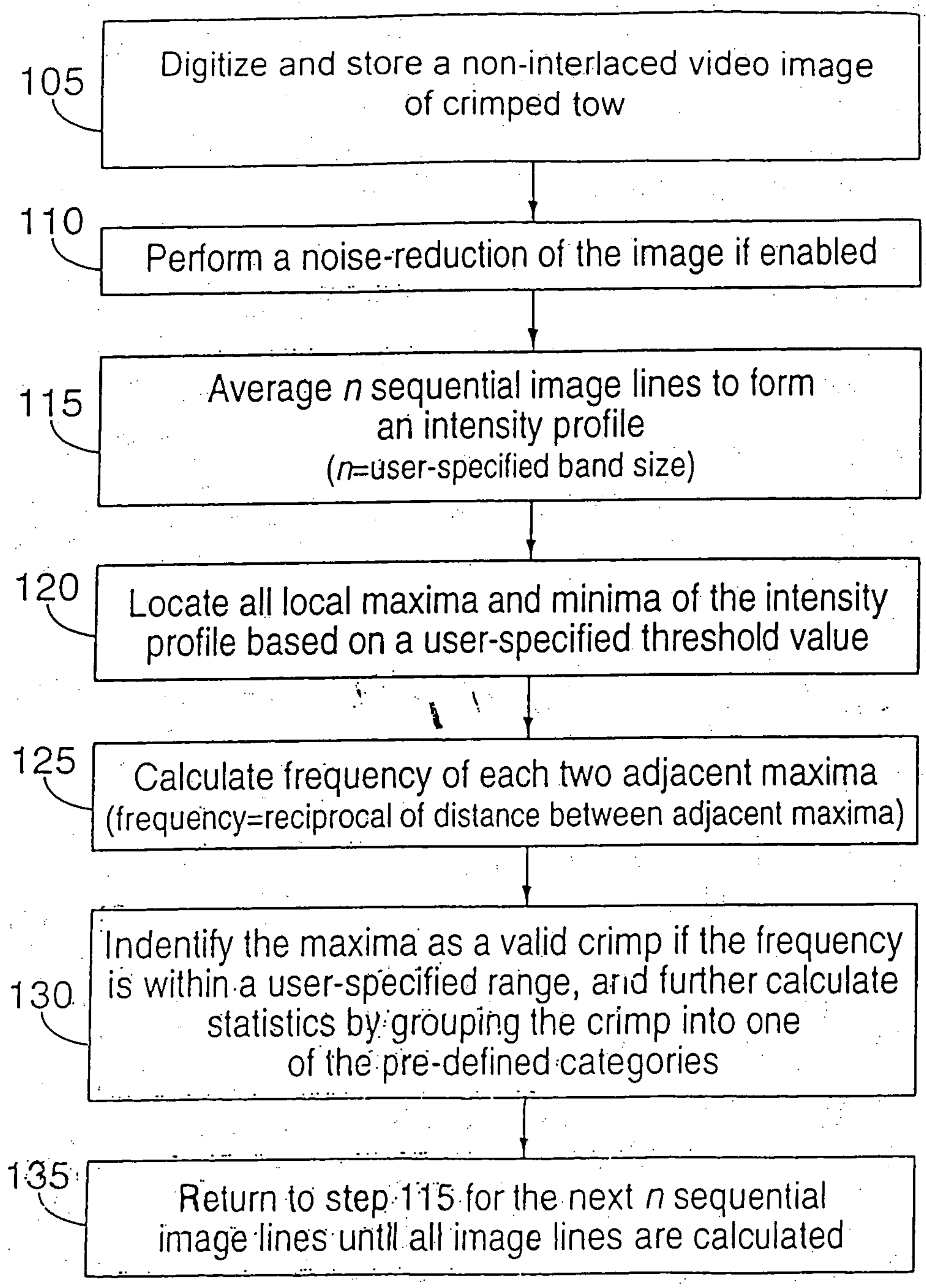

105
Digitize and store a non-interlaced video image of crimped tow
110
Perform a noise-reduction of the image if enabled
115
Average *n* sequential image lines to form an intensity profile
(*n*=user-specified band size)
120
Locate all local maxima and minima of the intensity profile based on a user-specified threshold value
125
Calculate frequency of each two adjacent maxima
(frequency=reciprocal of distance between adjacent maxima)
130
Indentify the maxima as a valid crimp if the frequency is within a user-specified range, and further calculate statistics by grouping the crimp into one of the pre-defined categories
135
Return to step 115 for the next *n* sequential image lines until all image lines are calculated

FIG. 2

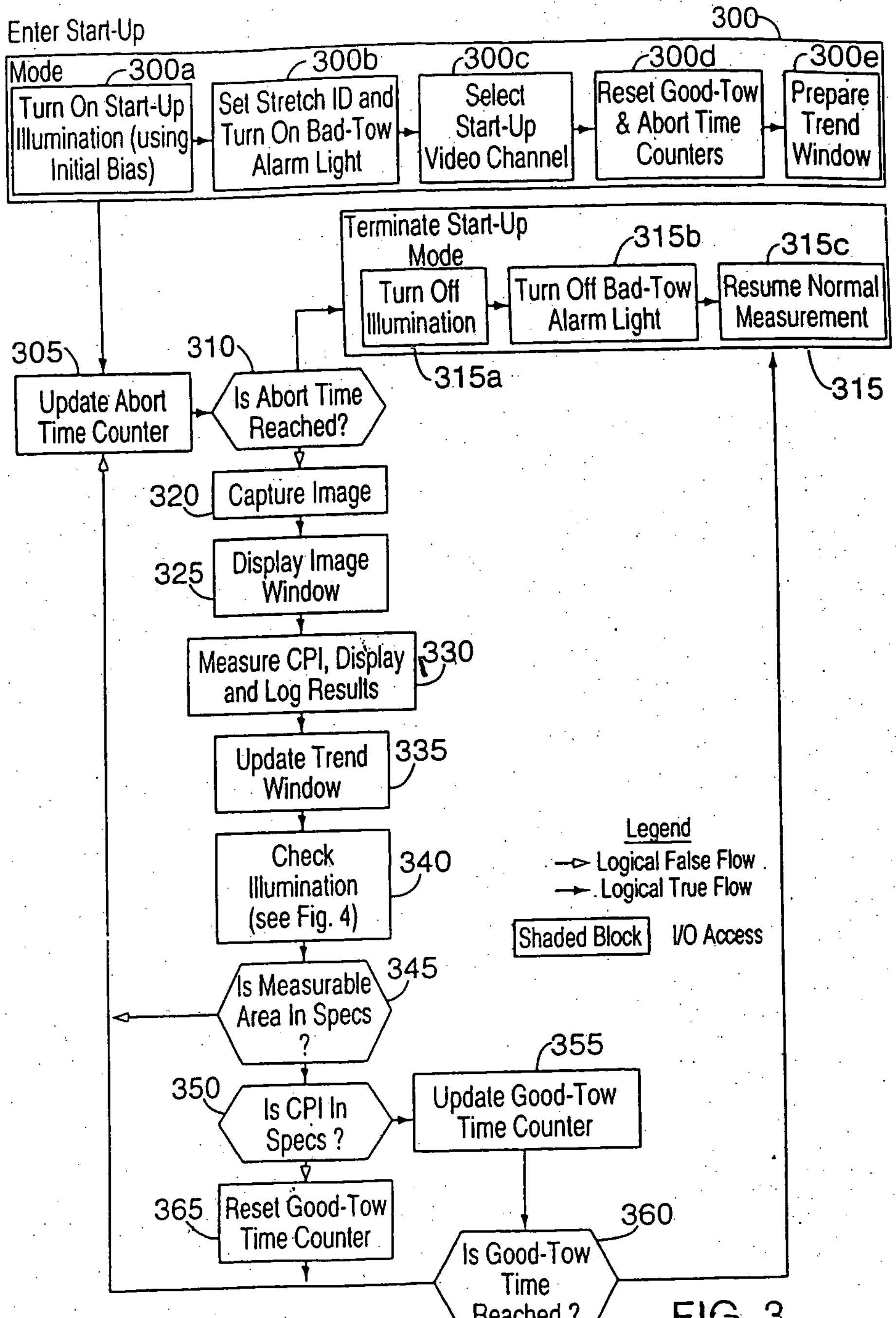

300
Enter Start-Up Mode
300a
Turn On Start-Up Illumination (using Initial Bias)
300b
Set Stretch ID and Turn On Bad-Tow Alarm Light
300c
Select Start-Up Video Channel
300d
Reset Good-Tow & Abort Time Counters
300e
Prepare Trend Window
Terminate Start-Up Mode
315a
Turn Off Illumination
315b
Turn Off Bad-Tow Alarm Light
315c
Resume Normal Measurement
315
305
Update Abort Time Counter
310
Is Abort Time Reached?
320
Capture Image
325
Display Image Window
330
Measure CPI, Display and Log Results
335
Update Trend Window
340
Check Illumination (see Fig. 4)
345
Is Measurable Area In Specs ?
350
Is CPI In Specs ?
355
Update Good-Tow Time Counter
365
Reset Good-Tow Time Counter
360
Is Good-Tow Time Reached ?
Legend
Logical False Flow
Logical True Flow
Shaded Block
I/O Access

FIG 3

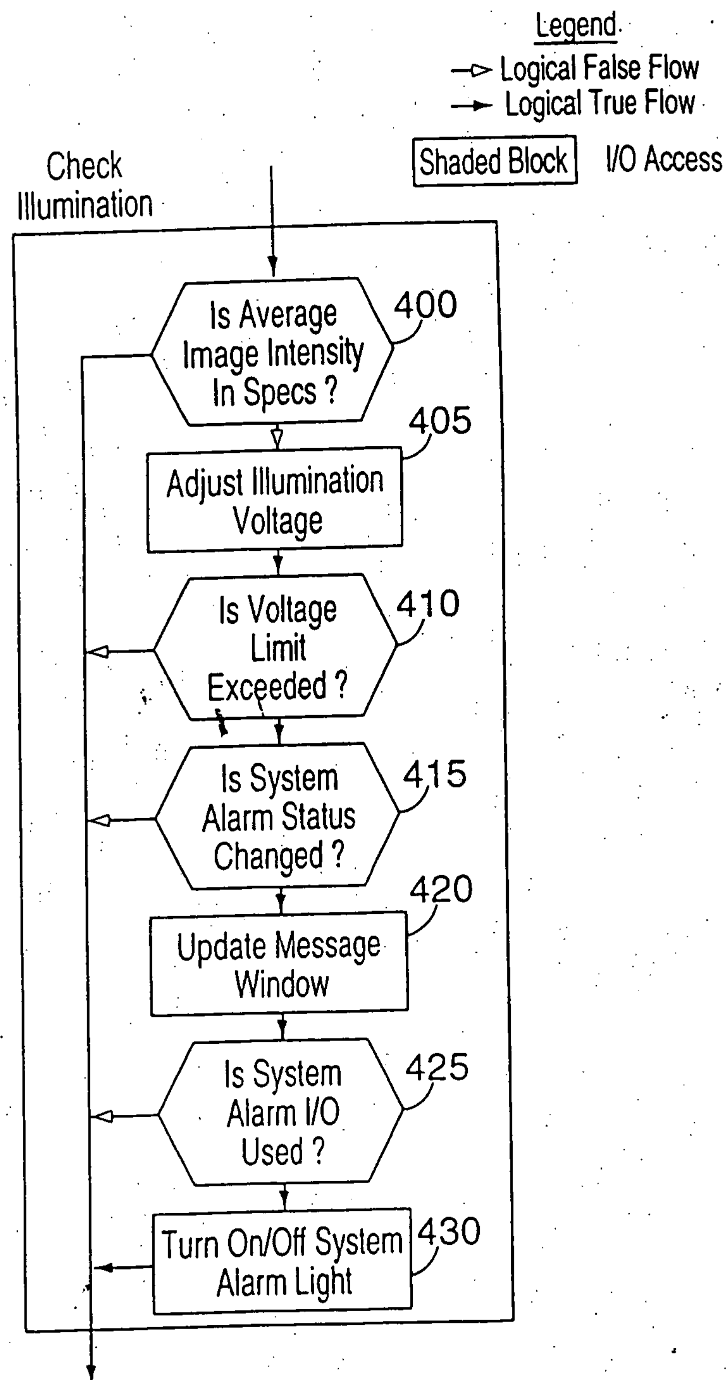
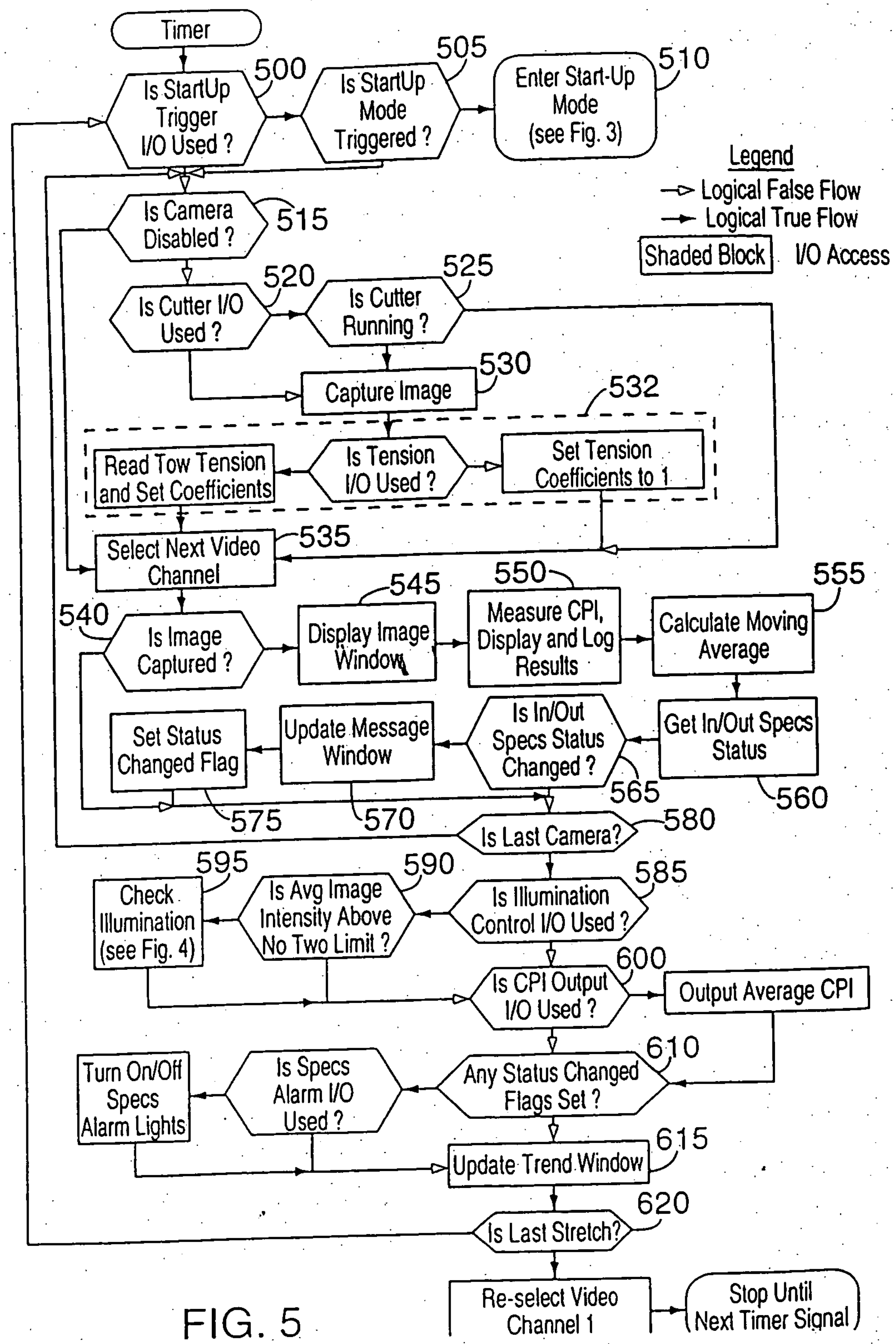
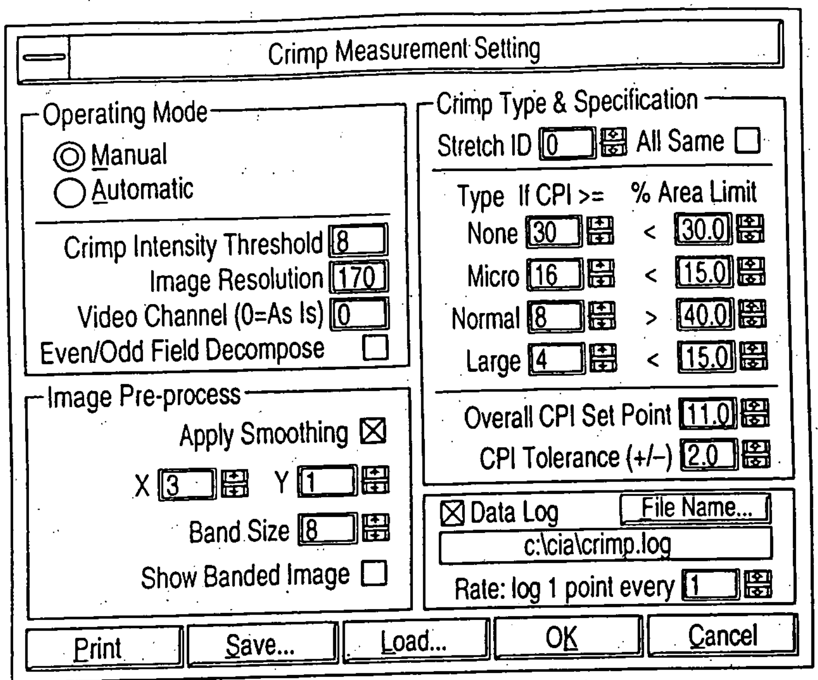

Legend
Logical False Flow
Logical True Flow
Shaded Block
I/O Access
Check Illumination
Is Average Image Intensity In Specs ?
400
Adjust Illumination Voltage
405
Is Voltage Limit Exceeded ?
410
Is System Alarm Status Changed ?
415
Update Message Window
420
Is System Alarm I/O Used ?
425
Turn On/Off System Alarm Light
430

FIG. 4

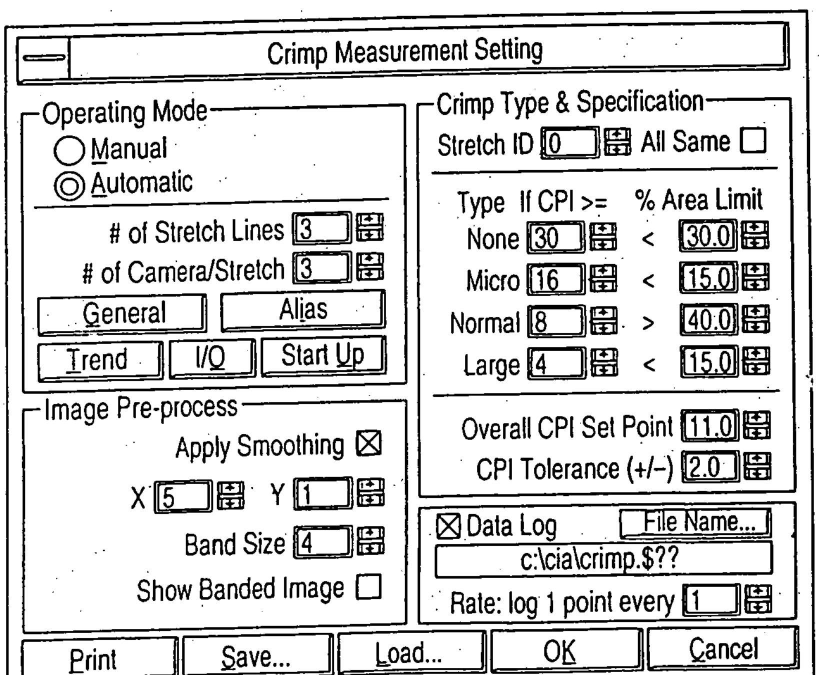


Timer
Is StartUp Trigger I/O Used ?
500
Is StartUp Mode Triggered ?
505
Enter Start-Up Mode (see Fig. 3)
510
Legend
Logical False Flow
Logical True Flow
Shaded Block
I/O Access
Is Camera Disabled ?
515
Is Cutter I/O Used ?
520
Is Cutter Running ?
525
Capture Image
530
532
Read Tow Tension and Set Coefficients
Is Tension I/O Used ?
Set Tension Coefficients to 1
Select Next Video Channel
535
540
Is Image Captured ?
Display Image Window
545
Measure CPI, Display and Log Results
550
Calculate Moving Average
555
Set Status Changed Flag
Update Message Window
Is In/Out Specs Status Changed ?
Get In/Out Specs Status
565
560
575
570
Is Last Camera?
580
Check Illumination (see Fig. 4)
595
Is Avg Image Intensity Above No Two Limit ?
590
Is Illumination Control I/O Used ?
585
Is CPI Output I/O Used ?
600
Output Average CPI
Turn On/Off Specs Alarm Lights
Is Specs Alarm I/O Used ?
Any Status Changed Flags Set ?
610
Update Trend Window
615
Is Last Stretch?
620
Re-select Video Channel 1
Stop Until Next Timer Signal

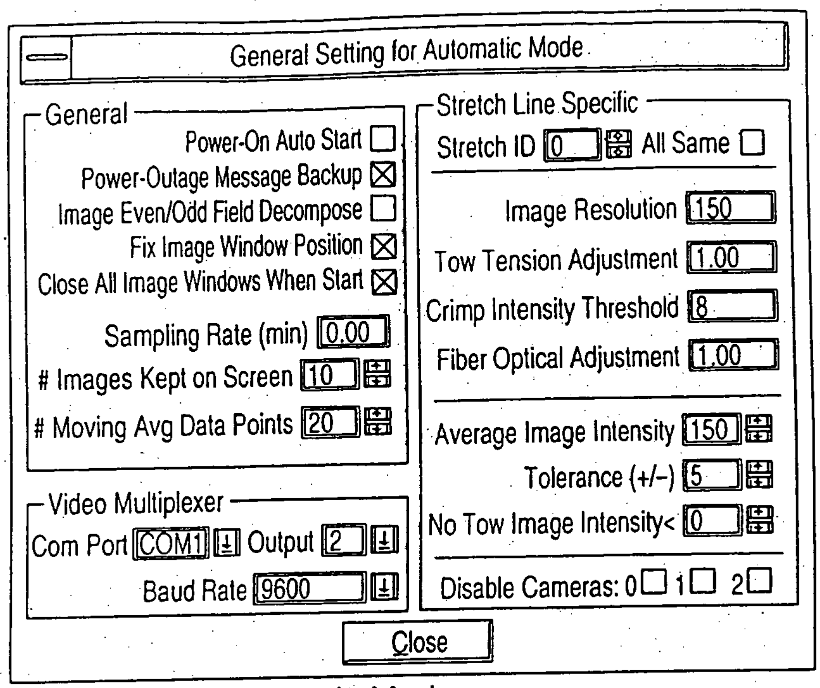
FIG. 5



Measurement Setting For Manual Mode FIG. 6A



Measurement Setting For Automatic Mode FIG 6B



'General' for Automatic Mode

FIG. 7A

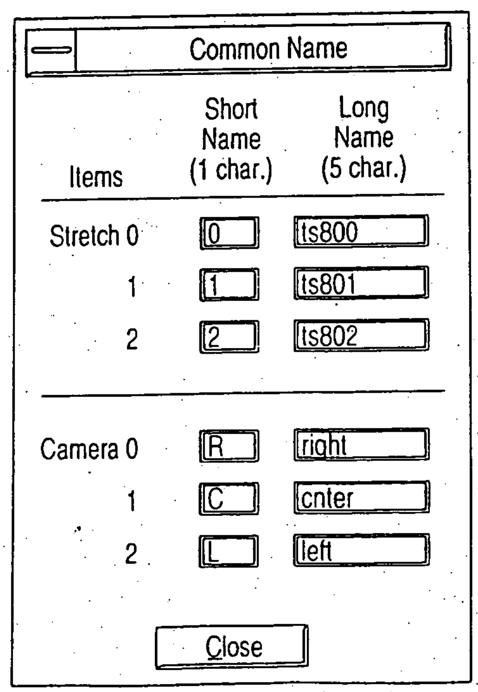
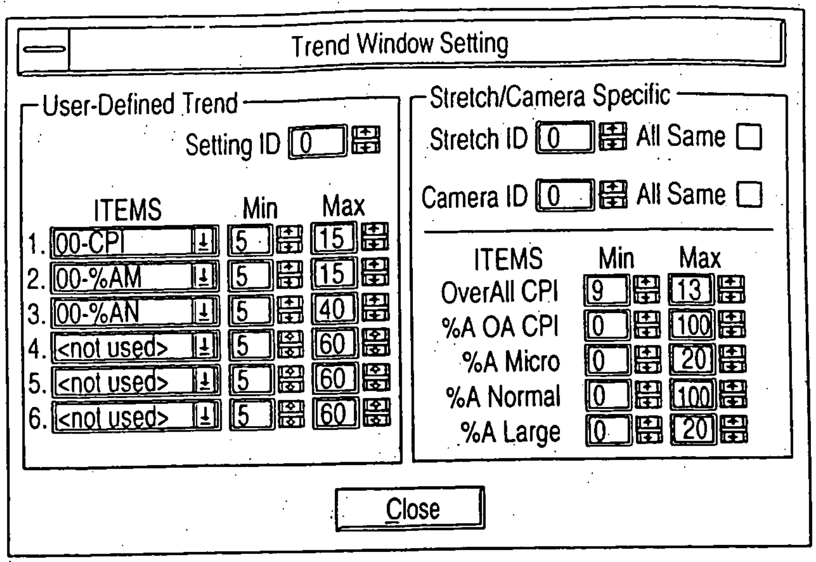
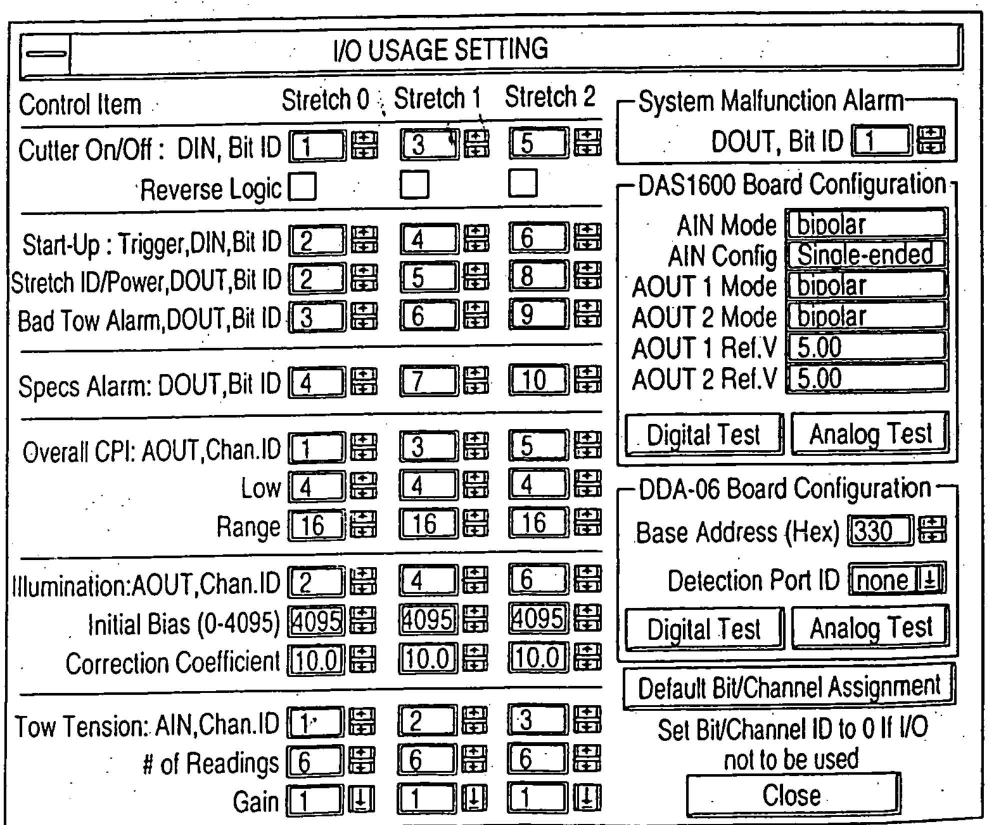


FIG. 7B



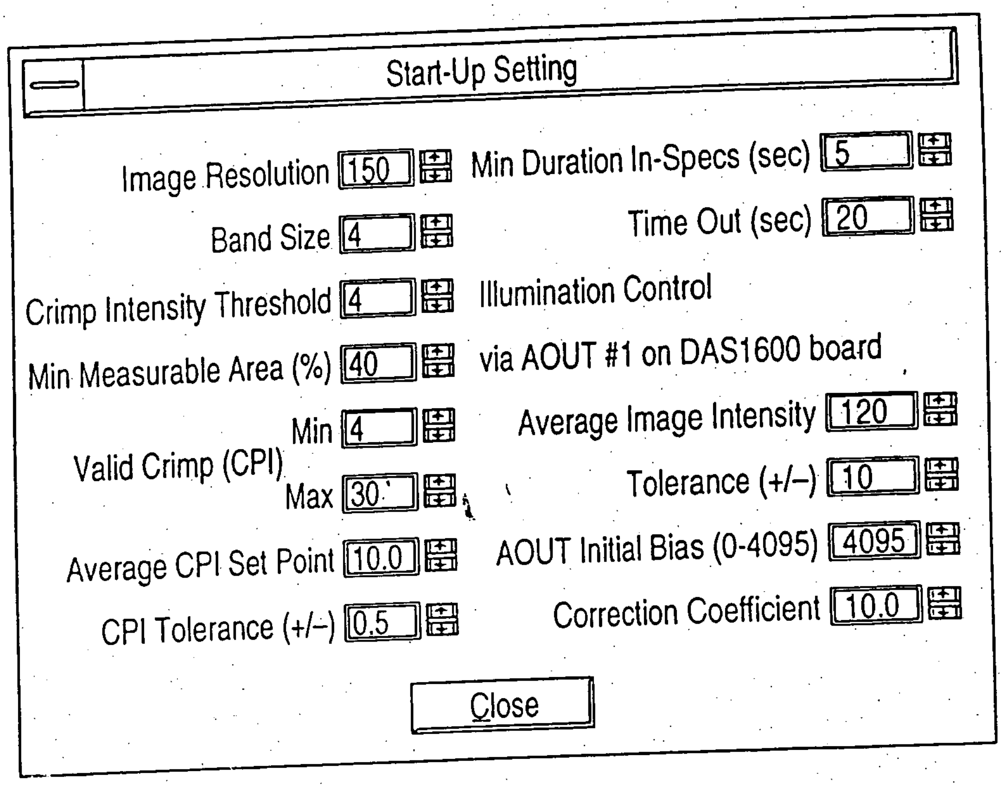
'Trend' for Automatic Mode

FIG. 7C



'I/O' for Automatic Mode

FIG 7D



'Start Up' for Automatic Mode

FIG. 7E

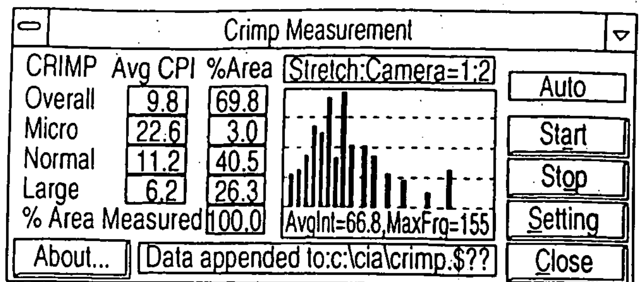


FIG. 8

Main Control Panel And Measurement Results Of The Current Image

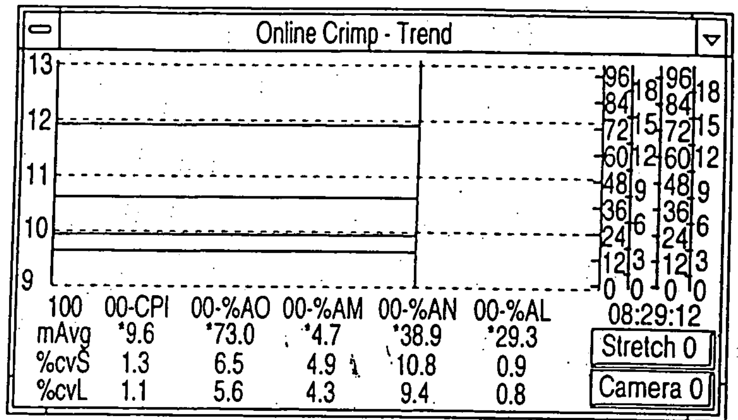


FIG. 9

Trend Window of Moving Average

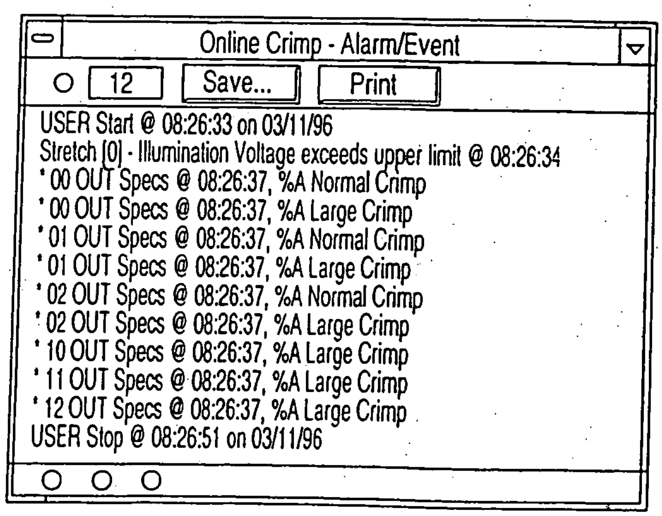
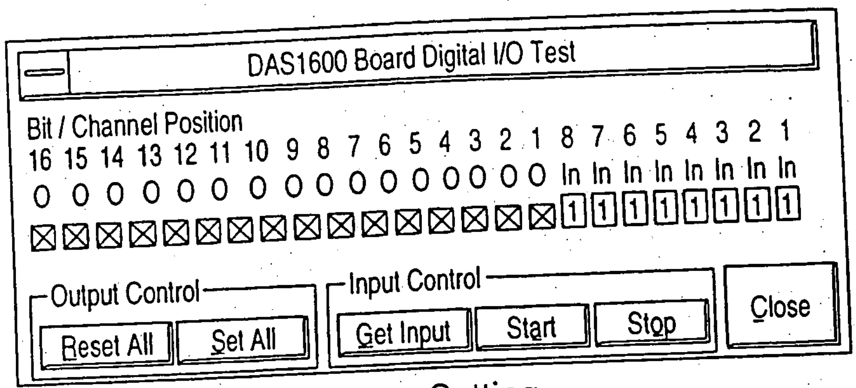


FIG. 10

Alarm/Event Message Window



'Digital Test' for I/O Usage Setting

FIG. 11A

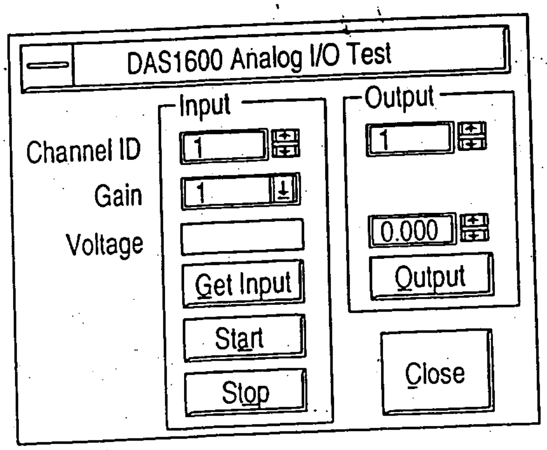
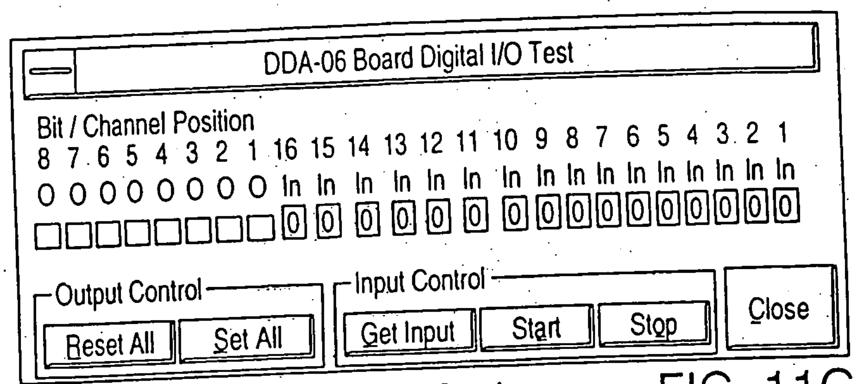


FIG. 11B

'Analog Test' for
I/O Usage Setting



'Digital Test' for I/O Usage Setting

FIG. 11C

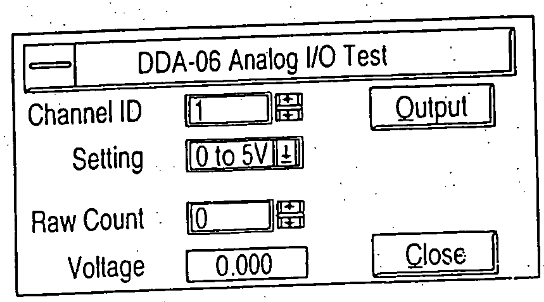


FIG. 11D

'Analog Test' for
I/O Usage Setting

FIG. 12A

```
/*-------------------------------------------------------------------------------
Measurement function activated by system's timer
-------------------------------------------------------------------------------*/
static void PNEAR NormalMeasurement(HWND hwnd)
{
    HANDLE hDIB[2];                         // handle to newly captured images
    LPIOUSAGE lpIO;                         // pt to IO setting data
    int  err=0, maCalc[2]={0,0};            // flag for error and moving avg calculation status
    int  s,c,idxm,idxm2,i,k;                // loop control variables
    float oaCPI[2];                         // avg overall cpi of a stretch
    int   nCPI[2];                          // # of images for avg overall cpi calculation
    float avgIntensity;                     // avg image intensity of a stretch
    int   nIntensity;                       // # of images for avg image intensity calculation
    extern LONG nUntitled;                  // # of image windows created since system started

    idxm=lpRes->IdxM+1;                             // get moving avg buffer idx
    for(s=0; s<lpCFG->nStretch; s++) {              // loop over 3 stretchers, actual # can be varied by user
        lpIO=&lpCFG->io[s];                         // get pt to io setting data
        if(lpIO->suTrig>=0 && ioIsStartup(lpIO->suTrig,s)) {   // check start-up mode trigger
            StartUpMode(hwnd,suENTER); return;
        }
        oaCPI[0]=oaCPI[1]=0.0f; nCPI[0]=nCPI[1]=0;      // init. variables of avg overall CPI
        avgIntensity=0.0f; nIntensity=0;                // init. variables of avg image intensity
        for(c=0; c<lpCFG->nCamera; c++) {               // loop over 3 cameras, actual # can be varied by user
            hDIB[0]=hDIB[1]=NULL;                       // initialize memory handle to NULL
            if(!lpCFG->disableCamera[s][c] &&           // camera not disabled
                (lpIO->cutter<0 ||                      // cutter I/O not used
                ioIsCutterOn(lpIO,s))) {                // or cutter is ON
                if(err=GetLiveImage(lpCFG->actype[s].dpi,hDIB)) goto EXIT;
                if(lpIO->tension>=0) ioGetTowTension(lpIO);
            }
            if(lpCtl->LastVideoCode!='2') {             // switch video channel if more than 1 camera used
                ...                                     // advance to next channel
            }
            for(i=0;i<nImgCap;i++) {                    // loop over 2 field-decomposed images
                if(hDIB[i]) {                           // image captured with GetLiveImage()
                    wsprintf(lpCtl->logName,cMg[73],s,c,cMg[39+i],nUntitled+1);
                    if(!ImageWindowAdd(hDIB[i],lpCtl->logName,1)) {    // create new image window
                        hDIB[i]=NULL; err=IDE_NoMemory; goto EXIT;     // fail to create new window
                    }
                    if(err=MeasureCrimpAuto(hwnd,s,c)) goto EXIT;      // measure crimp
                    if(MovingAvgGet(s,c,idxm2)) {       // calculate moving avg
```

FIG. 12B

```
                maCalc[i]++;                    // cumulate if moving average calculated
                oaCPI[i]+=lpMov[s][c]->pM[0][idxm2]; nCPI[i]++;
            }
            avgIntensity+=lpRes->avgIntensity;  // cumulate average image intensity for illumination control
            nIntensity++;
        }
    }   //--- end of loop over 2 images per capture
    ... // check user interrupts from mouse or keyboard
  }     //--- end of loop over cameras
  if(lpIO->illumin>=0 && nIntensity) {          // check illumination if I/O enabled
      avgIntensity/=(float)nIntensity;
      if(avgIntensity>=(float)lpCFG->LowInt[s]) ioLightingNormal(lpIO,s,avgIntensity);
  }
  if(lpIO->oaCPI>=0)                             // output overall avg CPI
      for(i=0;i<nImgCap;i++) if(nCPI[i]) ioOutputCPI(lpIO,oaCPI[i]/nCPI[i]);
  k=0;
  for(c=0;c<lpCFG->nCamera;c++)    // check/update measurement In/Out specs
      for(i=0;i<nITEMS;i++)        // loop over all cameras and measurement attributes
          if(lpAlm->msg[s][c][i]) { k=1; c=nCAMERA; break; }
  if(k!=lpAlm->curSpecWarn[s]) {   // if warning (alarm light) status changed
      ...                          // update status
  }
 }          //--- end of loop-over stretch
 if(maCalc[0]||maCalc[1]) {        // moving avg calculated for at least 1 stretch line
     ...                           // update trend window
 }
EXIT:
 if(err || InTimer==2) {           // Error stop or User stop
     StartStop(hwnd,0,!err);       // stop auto measurement first
     if(err) {                     // if error stop
        ...                        // error handling routines
     }
 }
}
/*-------------------------------------------------------------------------------------
GetLiveImage
-------------------------------------------------------------------------------------*/
int PFAR GetLiveImage(
int   dpi,      // image resolution, determined by camera optics and geometry
HANDLE *h)      // pt to array of handle to image data
{
    HANDLE hMem;
```

FIG. 12C

```
    int    err=IDE_NoMemory;
    if(lpCFG->DigitalOutput) *h=GetDigitalImage();      // get image data from camera digital output
    else if(hMem=TP_DataOnBoardGet(                     // get image data from frame grabber
        0,0,pBd->data.width-1,pBd->data.height-1)) {
        TGA2DIBmemBoard(hMem,dpi);                      // convert TGA to DIB format
        if(lpCFG->field) {                              // if field decompose required
            if(FieldDecompose(hMem,h)) err=0;           // no error
        } else { *h=hMem; err=0; }                      // if no decompose, output 1 handle
    }
    return(err);
}
/*--------------------------------------------------------------------------
Return: TRUE if OK, FALSE if run-out memory error
--------------------------------------------------------------------------*/
int PFAR FieldDecompose(HANDLE src,HANDLE *h)
{
    LPBITMAPINFOHEADER srclpbi=(LPBITMAPINFOHEADER)GlobalLock(src);
    LPBITMAPINFOHEADER dstlpbi[2];
    DWORD memSize, srcWidthByte=GetWidthByte(srclpbi);
    WORD  headSize=(WORD)srclpbi->biSize+(WORD)srclpbi->biClrUsed*sizeof(RGBQUAD);
    WORD  dy[2];
    BYTE _huge* s, _huge* d[2];
    int   i, k, rtn=TRUE;

    dy[0]=((WORD)srclpbi->biHeight+1)>>1;           // destination image1 height
    dy[1]= (WORD)srclpbi->biHeight-dy[0];           // destination image2 height
    h[0]=h[1]=NULL;
    for(i=0; i<2; i++) {                            // allocate memory buffers and copy image header data
        memSize=(DWORD)headSize+(DWORD)dy[i]*srcWidthByte;
        if(h[i]=GlobalAlloc(GMEM_MOVEABLE,memSize)) {
            dstlpbi[i]=(LPBITMAPINFOHEADER)GlobalLock(h[i]);
            _fmemcpy(dstlpbi[i],srclpbi,headSize);      // copy image head info
            dstlpbi[i]->biHeight=dy[i];
            dstlpbi[i]->biSizeImage=dstlpbi[i]->biHeight*srcWidthByte;
            d[i]=PointToData(dstlpbi[i]);
        } else rtn=FALSE;
    }
    if(rtn) {
        s=PointToData(srclpbi);                     // point to source image data
        k=(int)srclpbi->biHeight%2;                 // even/odd field index
        for(i=0; i<(int)srclpbi->biHeight; i++) {
            k=!k;                                   // change field index alternatively
```

FIG. 12D

```
        _fmemcpy(d[k],s,(WORD)srcWidthByte);// copy image data from source to destination
        d[k]+=srcWidthByte;                 // advance point to next image data row of destination image
        s   +=srcWidthByte;                 // advance point to next image data row of source image
      }
      GlobalUnlock(h[0]);
      GlobalUnlock(h[1]);
    } else if(h[0]) { GlobalUnlock(h[0]); GlobalFree(h[0]); h[0]=NULL; }
    GlobalUnlock(src);
    return(rtn);
}
/*-----------------------------------------------------------------------------
Return: 0 if OK, IDE_?? if Fail
-----------------------------------------------------------------------------*/
static int PNEAR MeasureCrimpAuto(
HWND hwnd,              // handle to caller's window
int sId,int cId)        // stretch and camera ID
{
    if(Pref.UndoEnable&&(PtActWnd->DIB2=DIBDupFull(PtActWnd->DIB))==NULL) return(IDE_NoMemory);
    lpRes->avgIntensity=TowEdgeDetection(PtActWnd->DIB,1);
    if(lpCFG->prep[1].smooth) {          // pre-process image if noise reduction is enabled
      LPBITMAPINFOHEADER lpbi=(LPBITMAPINFOHEADER)GlobalLock(PtActWnd->DIB);
      Filter(hwndStatus,0,lpbi,PtActWnd->DIB2,0,lpCFG->prep[1].x,lpCFG->prep[1].y,SMOOTH_AVERAGE,0,0,0.0f);
      GlobalUnlock(PtActWnd->DIB);
    }
    FindCrimp(PtActWnd->DIB,lpCFG->prep[1].bandsize,lpCFG->prep[1].showBand);  // identify/validate crimps
    if(lpCtl->nLogdata==1) return(WriteLog(sId,cId));   // log measurement result to a disk file
    return(0);
}
/*-----------------------------------------------------------------------------
Return: 0 if OK, IDE_?? if Fail
-----------------------------------------------------------------------------*/
static void PNEAR FindCrimp(
HANDLE memSrc,                 // src image to find crimp
int    bandsize,               // user-specified band size
char   showBand)               // user-specified show band-avged image option
{
    LPBITMAPINFOHEADER lpbi=(LPBITMAPINFOHEADER)GlobalLock(memSrc);
    LPINT  Loc=lpRes->Loc;     // pointer to pre-allocated memory buffer for storing location Info
    LPBYTE Pxl=lpRes->Pxl;     // pointer to pre-allocated memory buffer for storing pixel intensity of the profile
    DWORD  ByteWidth=GetWidthByte(lpbi);      // # of byte per image data row
    DWORD  bandByte =ByteWidth*bandsize;      // # of byte per band of image data
    int    Width=(int)lpbi->biWidth;          // image width in pixel
```

FIG. 12E

```
BYTE    _huge* srcD, _huge* d;                   // point to src image data
int     nBand, b;                                 // # of band to process
int     i, k, first, N, ext, cpi;                 // loop control variables
LONG    mArea,mCunt,nArea,nCunt,lArea,lCunt;     // area and counter for micro/normal/large crimp
LONG    tArea,tCunt;                              // total area and counter
register WORD pv;                                 // pixel value

for(i=0;i<cpiHighLimit;i++) lpRes->pHist[i]=0;           // init. distribution data buffer
mArea=nArea=lArea=mCunt=nCunt=lCunt=0L;                  // init. area and counter variables
if(lpRes->avgIntensity) {
    N=lpRes->top-lpRes->bottom;                          // # image rows, excluding background
    nBand=N/bandsize;                                    // # of band to process
    srcD=PointToData(lpbi)+ByteWidth*lpRes->bottom;      // point to src image data
} else { N=nBand=0; }                                    // black image, or all background
lpRes->edge=100.0f*(1.0f-(float)N/(float)lpbi->biHeight);
b=nBand;                                                 // # of bands to process
while(b--) {                                             // loop over bands
    for(i=0;i<Width;i++) {                               // calculate banded avg
        d=srcD+i; pv=(WORD)*d;
        for(k=1;k<bandsize;k++) { d+=ByteWidth; pv+=(WORD)*d; }
        Loc[i]=(int)(pv/bandsize);
        Pxl[i]=(BYTE)Loc[i];
        if(showBand) {
            d=srcD+i; pv=Pxl[i]; *d=(BYTE)pv;
            for(k=1;k<bandsize;k++) { d+=ByteWidth; *d=(BYTE)pv; }
        }
    }
    if((N=FindPeakValley(Loc,Width,&first))>2) {         // at least 2 points
        N=IdentifyPeak(Loc,Pxl,N,first,cPkInt)-1;        // -1 for not checking the last one
        for(i=0;i<N;i++) {
            ext=Loc[i+1]-Loc[i];                         // distance between adjacent peaks
            cpi=(int)(dpiAdj/(float)ext);                // convert to cpi unit
            if(cpi>=cpiLowLimit && cpi<cpiHighLimit) lpRes->pHist[cpi]+=1;
            if(     ext<=cNone) continue;                // not counted if too small
            else if(ext<=cMicr) { mArea+=ext; mCunt++; }   // micro crimp
            else if(ext<=cNorm) { nArea+=ext; nCunt++; }   // normal crimp
            else if(ext<=cLarg) { lArea+=ext; lCunt++; }   // large crimp
            else continue;                               // not counted if too large
            d=srcD+Loc[i];                               // init. image data pt to draw mark
            for(k=0; k<ext; k++) *d++=0xff;              // low horizontal line
            d=srcD+Loc[i];
            for(k=0; k<bandsize; k++) {                  // mark found crimp
```

FIG. 12F

```
                *d=0xff; *(d+ext)=0xff; d+=ByteWidth;
            }
        }
    }
    srcD+=bandByte;
}
if(mArea) lpRes->m[0]=dpiAdj*(float)mCunt/mArea; else lpRes->m[0]=0.0f;     // micro crimp  cpi
if(nArea) lpRes->n[0]=dpiAdj*(float)nCunt/nArea; else lpRes->n[0]=0.0f;     // normal crimp cpi
if(lArea) lpRes->l[0]=dpiAdj*(float)lCunt/lArea; else lpRes->l[0]=0.0f;     // large crimp  cpi
if(tArea=mArea+nArea+lArea) {                                 // total crimped area
    tCunt=mCunt+nCunt+lCunt;                                  // total crimp count
    lpRes->o[0]=dpiAdj*(float)tCunt/(float)tArea;             // overall CPI
} else lpRes->o[0]=0.0f;
if(tArea=(LONG)nBand*Width) {                                 // total image area excluding background area
    lpRes->m[1]=100.0f*(float)mArea/tArea;                    // %Area covered: micro
    lpRes->n[1]=100.0f*(float)nArea/tArea;                    // %Area covered: normal
    lpRes->l[1]=100.0f*(float)lArea/tArea;                    // %Area covered: large
} else { lpRes->m[1]=lpRes->n[1]=lpRes->l[1]=0.0f; }
lpRes->o[1]=lpRes->m[1]+lpRes->n[1]+lpRes->l[1];              // %Area covered: Overall
ShowResult(hwndCrimp);                                        // display result

'*---------------------------------------------------------------------------
Returns: # of peak/valley points found in the array
----------------------------------------------------------------------------*/
.nt PFAR FindPeakValley(
.nt loc[],       // input array, replaced with location idx of peak/valley points found upon return
.nt nIn,         // # of point in the array
.nt *VP1st)      // +/-  = the 1st peak-valley point is peak/valley

    register int old, new;
    int nEqu;                      // # of equal value points
    int nOut=0;                    // # of peak/valley point in the array
    int i, sign;
    old=loc[0]; nEqu=0;
    for(i=1; i<nIn; i++) {                        // find 1st peak/valley point
        if(loc[i]!=old) {
            sign=(loc[i]>old)?1:-1;               // initial slope sign
            *VP1st=-sign;
            loc[nOut++]=nEqu>>1;                  // location index of 1st peak/valley point
            break;                                // break-out search for 1st point
        } else nEqu++;
    }
```

FIG. 12G

```
    old=loc[i]; nEqu=0;
    if(i<nIn) {
        for(i=i+1; i<nIn; i++) {
            new=loc[i];
            if(new!=old) {
                if((new>old && sign<0) ||       // valley point
                    (new<old && sign>0)) {      // peak point
                    loc[nOut++]=i-1-(nEqu>>1);  // record this turning point
                    sign=-sign;
                }
                nEqu=0;
            } else nEqu++;
            old=new;
        }
        loc[nOut++]=(nIn-1)-(nEqu>>1);          // the last peak/valley point
    }
    return(nOut);
}
/*---------------------------------------------------------------------------
Identify crimp based on intensity criteria 'threshold'
Idx to crimp peak is returned via input peak/valley idx array 'loc[]'
---------------------------------------------------------------------------*/
int PFAR IdentifyPeak(
int  loc[],     // input peak/valley index array, return Peak idx array
BYTE pxl[],     // pixel intensity value array
int  N,         // # of peak/valley in array 'loc'
int  first,     // >0, 1st index in array 'loc' points to a peak
int  threshold) // intensity threshold value
{
    int i, outN=0;
    int C, L, R;                // current peak idx, left- & right-side valley idx
    int cPxl;                   // current peak pixel intensity
    int NoCompare=1;            // when previous peak is identified as NOT crimp peak
                                // higher one of the previous and current peaks should
                                // be used for identifying crimp peak

    i=(first>0) ? 2 : 1;        // 1st peak to be examined, 1st idx point to a peak if first>0
    L=loc[i-1];                 // idx to left-side valley
    if((N-i)%2) N--;            // the last location is peak which should NOT be checked
                                // because no right-side valley to be compared
    for(; i<N; i+=2) {
        if(NoCompare || pxl[C]<pxl[loc[i]]) C=loc[i];
```

FIG. 12H

```
        cPxl=(int)pxl[C]-threshold;
        R=loc[i+1];
        NoCompare=1;            // default to use new peak value @ next time peak identification
        if(cPxl>=(int)pxl[L]&&cPxl>=(int)pxl[R]) {      // crimp peak found
            loc[outN++]=C;              // record idx in output array
            L=R;                        // right-side valley becomes left-side valley for next peak
        } else {                        // crimp peak Not found
            if(pxl[R]<pxl[L]) L=R;      // right-side valley is lower, use it as left-side valley @ next time
            else NoCompare=0;           // left-side valley is lower, need compare for highest peak @ next time
        }
    }
    return(outN);
}
```